Modélisation de la sémantique formelle des langages de programmation en UML et OCL

Jacques Malenfant

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modélisation de la sémantique formelle des langages de programmation en UML et OCL

Jacques Malenfant

N° 4499

Juillet 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4499--FR

Modélisation de la sémantique formelle des langages de programmation en UML et OCL

Jacques Malenfant*

Thème 1 — Réseaux et systèmes
Projet Triskell

Rapport de recherche n° 4499 — Juillet 2002 — 19 pages

Résumé : Nous proposons une méthode systématique et pragmatique de modélisation de la sémantique des langages de programmation fondée sur la description en UML et OCL de sémantiques dénotationnelles. Cette approche se fonde sur la précision mathématique de la sémantique dénotationnelle de même que sur l'accessibilité et le rôle prépondérant des définitions en UML/OCL pour répondre à deux grandes questions : (1) Comment communiquer aux programmeurs une définition précise d'un langage de programmation tout en restant accessible ? et (2) Comment intégrer à la MDA des modèles de la sémantique des langages de programmation de manière cohérente et uniforme avec une démarche UML ? Dans le contexte de la MDA, de tels modèles permettront d'appuyer le développement logiciel par une génération automatique de code vers de multiples langages cibles (éventuellement spécifiques à des domaines, ou DSL) tenant compte explicitement de leurs différences sémantiques. Nous illustrons notre approche par une sémantique du langage de spécification de la qualité de service QML.

Mots-clés : langages de programmation, sémantique dénotationnelle, UML, OCL, architectures dérivées des modèles (MAD)

* également Université de Bretagne sud, laboratoire VALORIA.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Modeling the formal semantics of programming languages using UML and OCL

Abstract: We propose a systematic and pragmatic method to model the semantics of programming languages by describing denotaional semantics in UML and OCL. This approach founds itself in the mathematical precision of denotational semantics as well as in the accessibility of UML/OCL models in order to answer two central questions : (1) How can we communicate to programmers a precise definition of a programming languages while being accessible ? and (2) How can we integrate in the MDA models for the semantics of programming languages coherently and uniformely with a UML approach ? In the MDA context, such models will allow us to sustain software development with an automatic generation of code towards multiple target languages (possibly domain-specific) while explicitely taking into account their semantics distinctions. We illustrate our approach with a semantics of the QML language for the specification of quality of service contracts for software components.

Key-words: programming languages, denotational semantics, UML, OCL, Model-Driven Architectures

1 Introduction

Cet article propose une démarche systématique de modélisation de la sémantique des langages de programmation fondée sur la description en UML et OCL de sémantiques dénotationnelles. Par cette démarche, nous souhaitons répondre à deux grandes questions :

- Comment communiquer aux programmeurs une définition claire, précise et non-ambigüe d'un langage de programmation tout en restant accessible ?
- Comment intégrer à la MDA des modèles de la sémantique des langages de programmation de manière cohérente et uniforme avec une démarche UML ?

Les langages de programmation demeurent des technologies centrales du développement logiciel. Que ce soit en programmation «manuelle» ou par génération automatique de code, ils le resteront encore longtemps dans la mesure où ils permettent d'exprimer finement les détails d'une application qu'il n'est pas souhaitable de représenter directement dans les modèles. Pourtant, l'utilisation des langages reste fragilisée par une compréhension approximative de leur sémantique précise par les programmeurs. Déjà redoutée pour les langages généralistes, ce problème prend une dimension plus importante avec l'avènement des langages spécifiques («*domain-specific languages*»).

Ces «petits» langages, qui peuvent prendre des formes diverses, d'une DTD XML à une syntaxe pur cru, jouent un rôle de plus en plus important dans le développement logiciel. Il est essentiel de pouvoir se les approprier dans le détail. La sémantique formelle propose une approche de définition claire et non-ambigüe, mais elle souffre d'une coupure par rapport aux connaissances de la majorité des programmeurs. L'alternative que nous proposons ici consiste à exprimer l'essence de la sémantique formelle sous la forme d'un modèle UML. Nous ne prétendons pas, bien sûr, que le modèle ainsi obtenu préserve toutes les propriétés de la sémantique, mais plutôt que les deux définitions présenteront conjointement les avantages de formalité et de compréhensibilité.

Dans le contexte de la MDA, l'intégration de tels modèles de langages permettra d'appuyer le développement logiciel sur une explicitation précise des liens entre modèles et programmes. Le développement logiciel piloté par les modèles se fonde sur une génération de code vers de multiples langages cibles, que ce soit pour différentes versions du logiciel ou encore pour différents aspects d'une même application. Par l'expression de la sémantique des langages utilisés sous forme de modèles manipulables en amont de la génération, la modélisation et la génération peuvent tenir compte explicitement des différences entre les sémantiques des langages généralistes (C++ versus Java, par exemple) ou encore du potentiel des langages spécifiques. Ceci est à contraster avec une approche où les générateurs de code sont externes au modèle, et donc des boîtes noires du point de vue modélisation.

Dans le présent article, nous avons choisi de nous fonder sur les sémantiques dénotationnelles. Ce choix n'est pas limitatif de l'approche. À l'évidence, les sémantiques axiomatiques peuvent être également utilisées, comme l'illustre la définition de l'*Action Semantics* d'UML où la sémantique de chaque construction est définie en terme de pré- et postconditions écrites en OCL [OMG01]. Outre le fait de se donner une alternative à cette approche axiomatique, et sans vouloir entrer dans une polémique entre différentes approches formelles, la sémantique dénotationnelle présente de notre point de vue suffisamment d'avantages pour justifier son apport dans le contexte de la MDA. Pour répondre aux deux questions précédentes, notre approche se fonde donc sur la précision mathématique de la sémantique dénotationnelle et sur l'accessibilité et le rôle prépondérant des définitions en UML, s'imposant comme *lingua franca* de l'approche MDA.

Dans le reste de cet article, nous montrons comment passer d'une sémantique dénotationnelle à un modèle UML/OCL par une méthode systématique et pragmatique. Notre approche s'apparente au patron de conception *Interpréteur*, à la base de tous les processeurs de langages implantés selon l'approche objet et qui a été présenté dans l'ouvrage fameux de la bande des quatre [GHJV95, p. 243]. Il s'agit donc d'adapter cette idée générale au cas dénotationnel, en prenant en compte ses spécificités qui sont introduites de manière pédagogique dans la première partie de l'article. Après la définition du modèle syntaxique du langage, nous montrons comment structurer le modèle sémantique d'abord par une traduction systématique des domaines et valeurs sémantiques puis

```
type Reliability = contract {
  failureMasking : decreasing set { omission, lostResponse, noExecution,
                                    response, responseValue, stateTransition } ;
  serverFailure : enum { halt, initialState, rolledBack } ;
  operationSemantics : decreasing enum { atLeastOnce, atMostOnce, once }
                  with order { once < atLeastOnce, once < atMostOnce } ;
  rebindingPolicy : decreasing enum { rebind, noRebind }
                  with order { noRebind < rebind } ;
  dataPolicy : decreasing enum { valid, invalid } with order { valid < invalid } ;
  numberOfFailures : decreasing numeric no / year ;
  availability : increasing numeric  ;
} ;

systemReliability = Reliability contract {
  failureMasking < { noExecution, response } ;
  serverFailure == initialState ;
  operationSemantics <= { atMostOnce } ;
  rebindingPolicy < rebind ;
  dataPolicy < invalid ;
  numberOfFailures < 10 no / year ;
  availability > 0.8 ;
} ;
```

FIG. 1 – Exemple de contrat en QML

en rattachant la sémantique statique par des préconditions et des invariants, et la sémantique dynamique par des postconditions sur les opérations définies par le modèle.

Exemple. Le langage QML. Nous illustrons notre approche par une sémantique d'une partie du langage QML (*QoS Modeling Language*) de spécification déclarative de la qualité de service des composants logiciels que nous avons réalisée précédemment [Mal02]. Un programme QML est essentiellement une série de déclarations de types de contrats, de contrats et de profils. Les profils, servant à associer contrats et interfaces, ne sont pas considérés ici. Un exemple de programme QML apparaît à la figure 1.

Un type de contrats est constitué d'une séquence de déclarations de dimensions, lesquelles sont formées d'un identificateur de dimension et d'un type de dimension (i.e. une sorte de dimension suivie ou non d'une unité de mesure). QML prévoit cinq sortes de dimensions. Les énumérations, énumérations avec ordre, ensembles et ensembles avec ordre permettent de définir des niveaux de qualité de service à l'aide d'identificateurs (i.e. modélisation *qualitative* de la qualité de service). Ces identificateurs définissent des valeurs de qualité de service, éventuellement comparables par la relation d'ordre. Dans les dimensions ensemblistes, des ensembles de valeurs sont admis entre lesquels une relation d'ordre est établie (inclusion si les valeurs individuelles ne sont pas comparables, inclusion avec majorant sinon). La dernière sorte de dimension est la dimension numérique, servant bien entendu à la modélisation quantitative de la qualité de service. Une information importante donnée par une déclaration de dimension est la sémantique ou signification de la notion de valeur préférable ou meilleure valeur (`increasing` ou `decreasing`).

Un contrat pour sa part est défini par rapport à son type de contrat, désigné directement par son identificateur dans les déclarations simples et indirectement par le contrat étendu dans les contrats définis par raffinement d'un contrat pré-existant. Outre cela, une définition de contrat est constituée d'une série de contraintes sur les dimensions déclarées dans le type de contrat. Une contrainte applique aux valeurs mesurées selon la dimension un opérateur de comparaison classique et une valeur cible de la contrainte. □

2 La sémantique dénotationnelle

2.1 Principes généraux

Du point de vue de la sémantique, la définition d'un langage débute par une syntaxe abstraite. Cette syntaxe abstraite exprime toutes les phrases du langage sans s'embarrasser des détails qui concernent le rendu physique d'un programme sous la forme d'une séquence de caractères analysable selon des méthodes d'analyses lexicale et syntaxique connues. La sémantique vise, pour l'essentiel, à définir comment attribuer une signification à chacune des phrases des programmes, en s'appuyant sur la syntaxe abstraite du langage. Une sémantique formelle s'attache à prendre comme éléments de signification des objets mathématiques formellement manipulables. Le choix des objets mathématiques utilisés pour ce faire est ce qui distingue les différents types de sémantique.

La sémantique dénotationnelle doit son nom au fait qu'elle voit les phrases de la syntaxe comme «dénotant» l'objet mathématique explicitant leur signification. La sémantique dénotationnelle repose sur deux grands principes : la compositionalité et le calcul de points fixes. La compositionalité est la propriété d'une sémantique dont la signification d'une phrase ne dépend que de la signification de ses sous-phrases. Plus techniquement, on pourrait dire que les objets mathématiques représentant la signification d'une phrase ne sont que le résultat de la composition des objets mathématiques représentant la signification de ses sous-phrases. Cette propriété est importante dans la mesure où elle permet de faire des preuves de propriétés sur les programmes par induction structurelle sur ses phrases, en utilisant des règles de déduction définies sur les différents types de phrases de la syntaxe abstraite.

Tout langage de programmation utilise des structures «auto-référentielles», comme les itérations, la récursivité ou encore le «*self*» des langages à objets. L'objectif de donner à ces structures une signification sous la forme d'un objet mathématique et la contrainte de compositionalité a amené, pour la sémantique dénotationnelle, au choix d'obtenir la signification de ces structures auto-référentielles par un calcul de point fixe.

Considérons l'itération à titre d'exemple. Une sémantique *opérationnelle* dira que l'exécution d'une itération consiste à vérifier la condition de l'itération dans l'état courant, et si la condition est vraie, alors de calculer l'état résultant de la prochaine itération pour appliquer à nouveau la sémantique de l'itération à ce nouvel état. Une telle définition n'est pas compositionnelle, puisque la sémantique de l'itération dépend de sa propre sémantique (de manière récursive). La compositionalité de la sémantique dénotationnelle impose que la sémantique de l'itération ne s'exprime qu'à partir de la sémantique de la condition et de celle du corps de l'itération. L'idée consiste à définir la sémantique de l'itération comme une fonction d'un état initial vers un état final, et de l'obtenir par point fixe. On commence par définir cette fonction pour tous les états sur lesquels la condition est fausse, puisque sur ces états elle est l'identité. On passe ensuite aux états pour lesquels la condition est vraie et qu'une itération amène dans un état pour lequel la condition est fausse, et ainsi de suite avec ceux qui demandent deux, trois, ... itérations.

Il est important de constater dans cet exemple que la notion d'ordre utilisée pour l'obtention du point fixe est intimement liée à la quantité d'information connue sur le valeur à calculer. Le calcul de point fixe augmente à chaque étape la quantité d'information connue sur la fonction en ajoutant de nouveaux états pour lesquels elle est définie. L'ensemble des fonctions partielles définies sur les états nécessitant 0, 1, 2, ... itérations est borné par un point fixe qui est la fonction définie pour tous les états possibles à l'entrée de l'itération. C'est ce point fixe qui est choisi comme dénotation de l'itération.

L'obtention de points fixes étant fondamentale[1], il importe d'assurer que ceux-ci existent toujours. Pour cela, il faut choisir des objets mathématiques ayant de bonnes propriétés. Il est bien sûr possible d'exiger qu'ils forment un treillis complet, mais les travaux de Scott [Sco76, Sco82, Sto77] ont montré que cette exigence peut être assouplie en ordres partiels complets, ce qui a donné lieu à la définition de la notion de domaines. Le développement d'une théorie des domaines supporte aujourd'hui la presque totalité des travaux sur la sémantique dénotationnelle. D'autres structures

[1] Elle fut d'ailleurs appelée par certains «sémantique de point fixe (*fixpoint semantics*) à ses débuts.

mathématiques peuvent également se prêter au jeu, comme les espaces métriques dont les espaces de Banach [dBdV96], bien connus en recherche opérationnelle par exemple.

2.2 Constituents d'une sémantique dénotationnelle

La maturation de l'approche dénotationnelle a amené plusieurs auteurs à proposer ce que l'on peut appeler des présentations structurées de sémantiques dénotationnelles [Sch86, Mos90]. En gros, on distingue :

- la *syntaxe abstraite*, qui définit la structure de toutes les phrases sur lesquelles la sémantique doit travailler ;
- l'*algèbre sémantique*, qui définit l'ensemble des domaines de valeurs des objets mathématiques qui vont servir de dénotations pour les phrases de la syntaxe abstraite ;
- les *fonctions de valuations*, à raison d'une fonction par type de phrases de la syntaxe abstraite, qui vont définir par des équations quelle valeur du domaine sémantique dénote chacune des phrases abstraite d'un programme.

2.2.1 Syntaxe abstraite

Nous avons déjà évoqué la syntaxe abstraite, qui est un concept relativement bien connu des informaticiens (bien que souvent confondu avec la syntaxe concrète par les programmeurs). Une fois analysé, le programme peut révéler sa structure sous la forme d'une arborescence des phrases où n'apparaissent que les éléments porteurs de sens. Réduite à la définition d'arborescences, la syntaxe abstraite utilise généralement des grammaires.[2] Dans ces grammaires, les non-terminaux définissent les types de phrases, généralement appelés *domaines syntaxiques* en référence aux *domaines sémantiques*, et correspondent à des types de nœuds pouvant apparaître dans l'arbre de syntaxe abstraite. De la même manière que chaque type de phrases de la grammaire se décomposent en alternatives, un type de nœud dans un arbre de syntaxe abstraite a généralement plusieurs formes. Dans un langage de programmation fonctionnelle, un type de nœud serait défini comme un type somme.

Une syntaxe abstraite est spécifiée par la définition des domaines syntaxiques et la grammaire abstraite. La définition d'un domaine syntaxique lui attribue un nom et associe à ce nom une variable syntaxique servant à désigner les valeurs de ce domaine dans les productions subséquentes.
Exemple. Syntaxe abstraite de QML. La figure 2 présente la syntaxe abstraite partielle du langage QML. Un programme p est une séquence de déclarations de types de contrats et de contrats, suivie d'une expression de test t. Cette expression de test, ajouté pour la mise au point, vérifie si une valeur dv respecte la contrainte posée sur une dimension d'un contrat donné. Une déclaration de type de contrat ctd associe un identificateur i à un type de contrat ct, lequel est constitué d'une séquence de dimensions. Une dimension dim déclare une sorte de dimension ds qualifiée ou non par une unité de mesure. Les cinq sortes de dimension se retrouvent, avec une sémantique de relation rs s'il y a relation d'ordre, une séquence d'identificateurs jouant le rôle de valeurs, et une relation d'ordre o le cas échéant. Une relation d'ordre est constitué de relations rel entre valeurs deux à deux. Une déclaration de contrat associe un identificateur à une expression de contrat ce. L'expression de contrat peut être simple, et définir un nouveau contrat d'identificateur i et de contraintes co^+ ou encore le raffinement d'un contrat existant d'identificateur i auquel sont ajoutées les contraintes co^+. Une contrainte pose l'opérateur de contrainte cp et la valeur cible dv sur une dimension d'identificateur i. Les littéraux lit sont des identificateurs, des séquences d'identificateurs (ensembles de valeurs), et des réels pour les dimensions numériques. □

2.2.2 Algèbre sémantique

L'algèbre sémantique introduit tous les types d'objets mathématiques qui vont exprimer la signification des programmes. Comme nous l'avons vu, ces valeurs font partie de structures mathématiques aptes à supporter le calcul de points fixes. Le plus souvent, ce sont des domaines au

[2] Elles aussi abstraites, par rapport aux grammaires concrètes des syntaxes concrètes.

Domaines syntaxiques :

p	$\in$	$Program$	$unit$	$\in$	$Unit$
d	$\in$	$Decl$	rs	$\in$	$RelSem$
t	$\in$	$Test$	cd	$\in$	$ConDecl$
ctd	$\in$	$ConTypeDecl$	ce	$\in$	$ConExp$
ct	$\in$	$ConType$	co	$\in$	$Constraint$
dim	$\in$	$Dimension$	dv	$\in$	$DimValue$
dt	$\in$	$DimType$	lit	$\in$	$Literal$
ds	$\in$	$DimSort$	cp	$\in$	$ConstraintOp$
rel	$\in$	$Relation$	num	$\in$	$Number$
o	$\in$	$Order$	i	$\in$	$Identifier$

p ::= `(program` d^+ t `)`

d ::= `(decl-type` ctd `)` | `(decl-contract` cd `)`

t ::= `(test-verify` i i dv `)`

ctd ::= `(contract-type-decl` i ct `)`

ct ::= `(contract-type` dim^+ `)`

dim ::= `(dimension` i dt `)`

dt ::= `(dimtype-simple` ds `)` | `(dimtype-qualified` ds $unit$ `)`

ds ::= `(dimsort-enum` i^+ `)` | `(dimsort-enum-with` rs i^+ o `)` | `(dimsort-set` rs i^+ `)` | `(dimsort-poset` rs i^+ o `)` | `(dimsort-numeric` rs `)`

rel ::= `(relation` i i `)`

o ::= `(order` rel^+ `)`

$unit$ ::= `unit-percent` | `(unit-named` i^+ `)`

rs ::= `decreasing` | `increasing`

cd ::= `(contract-decl` i ce `)`

ce ::= `(conexp-simple` i co^+ `)` | `(conexp-refinement` i co^+ `)`

co ::= `(constraint-simple` i cp dv `)`

dv ::= `(dimvalue-qualified` lit $unit$ `)` | `(dimvalue-pure` lit `)`

lit ::= `(lit-simple` i `)` | `(lit-list` i^+ `)` | `(lit-num` num `)`

cp ::= `equal` | `gtequal` | `ltequal` | `gt` | `lt`

FIG. 2 – Domaines syntaxiques pour le langage QML

sens de Scott, c'est-à-dire des ensembles partiellement ordonnés. Il ne s'agit pas ici d'entrer dans le détail de la théorie des domaines. En fait, les principaux domaines utiles dans les langages de programmation courants ont été définis depuis belle lurette. D'ailleurs, l'un des aspects du bon développement de sémantiques dénotationelles, comme en développement logiciel, est de réutiliser les morceaux déjà définis. Il en va de même des domaines en sémantique dénotationnelle.

Premier élément de structuration, un grand nombre de domaines de base sont prédéfinis, en ce sens où leurs bonnes propriétes mathématiques ont été prouvées de longue date. C'est le cas par exemple des domaines dits discrets. Un domaine discret est un domaine constitué d'un ensemble de valeurs non-comparables entre elles au sens de la quantité d'information apportée par la valeur, par exemple les entiers, auquel est ajouté une valeur «absence d'information», notée $\perp$ (prononcez «*bottom*»). En terme de point fixe, il y a simplement deux possibilités, la première où la valeur n'est pas connue (i.e. $\perp$) puis la suivante où la valeur est connue (par exemple, 3).

À ces domaines simples s'ajoutent des opérateurs de construction de domaines dont on a prouvé que lorsqu'ils sont appliqués à des domaines, il donnent à nouveau un domaine. C'est le cas des principales structures de données classiques : produits (i.e. struct à la C), sommes (i.e.

- $d_1 \to d_2$ dénote le domaine des fonctions continues du domaine dénoté par d_1 vers le domaine dénoté par d_2. On a $f \sqsubseteq_{d_1 \to d_2} g$ ssi $\forall x \in d_1 . f(x) \sqsubseteq_{d_2} g(x)$.
- $d_1 \times d_2 \ldots \times d_n$ dénote le produit cartésien des domaines d_1, d_2, ..., d_n, pour tout $n \geq 2$. On a $(x_1, \ldots, x_n) \sqsubseteq_{d_1 \times d_2 \ldots \times d_n} (y_1, \ldots, y_n)$ ssi $x_i \sqsubseteq_{d_i} y_i$ pour $i = 1, \ldots, n$.
- $d_1 \otimes d_2 \ldots \otimes d_n$ dénote le produit cartésien discrétisant telle que tout n-tuple contenant une valeur $\perp_{d_i}$ est identifié à $\perp_{d_1 \otimes d_2 \ldots \otimes d_n}$. Ainsi, le produit cartésien discrétisant de domaines discrets est lui-même discret.
- $d_1 + d_2 \ldots + d_n$ dénote la sommes séparée de domaines d comprenant toutes les valeurs (distinguables) des domaines de la somme, auxquelles est adjoint un nouvel élément $\perp_{d_1 + d_2 \ldots + d_n}$. Les éléments de d provenant des différents domaines d_i sont incomparables dans d.
- $d_1 \oplus d_2 \ldots \oplus d_n$ dénote la somme unifiée des domaines où toutes les valeurs $\perp_{d_i}$ sont unifiées à la valeur $\perp_{d_1 \oplus d_2 \ldots \oplus d_n}$.
- d^* dénote le domaine des listes de longueur finie composées d'éléments dans le domaine d sauf $\perp_d$. Les listes de longueurs différentes sont incomparables par $\sqsubseteq$.

FIG. 3 – Algèbre des domaines

$(e_1\ e_2)$	application de $f : d \to d'$ dénotée par e_1 sur la valeur $x \in d$ dénotée par e_2.
$e_1 \circ e_2$	composition de fonctions.
$\mathbf{fix}_d$	opérateur de point fixe pour le domaine d.
$on_i(x)$	projection de la valeur x d'un domaine produit dans son $i$ème composante.
$x \mid_{\mathbf{D}}$	projection de la valeur x d'un domaine somme dans sa composante $\mathbf{D}$.
$in\mathbf{D}(x)$	injection de la valeur x dans le domaine somme $\mathbf{D}$.
$\langle \ldots \rangle$	construction de liste.
$l \downarrow k$	k^{e} membre de la liste l (partant de 1).
$\#l$	longueur de la liste l.
$l \S t$	concaténation des listes l et t.

FIG. 4 – Résumé de la notation utilisée

types sommes de CAML), listes, ... La figure 3, inspirée de Mosses [Mos90], présente partiellement l'algèbre de construction des domaines.

Une algèbre sémantique comprend la définition des domaines et la définition des opérations pouvant être appliquées sur ces valeurs, opérations qui seront considérées comme des «primitives» par les équations sémantiques qui vont suivre. Encore une fois, pour les domaines de base, les opérations standards ont depuis longtemps été définies. Pour les domaines construits selon l'algèbre des domaines, les opérations usuelles ont également été définies. C'est le cas par exemple de l'accès aux champs d'un produit, à l'extraction d'une valeur d'un domaine somme ou encore de la décomposition d'un liste en son premier élément et la suite de la liste (réminescence de `car` et `cdr`) (voir notations standards en figure 4). Il est donc plus rare de voir ces définitions de fonctions apparaître explicitement dans une sémantique dénotationnelle.

Exemple. Une sémantique, peu importe sa notation, cherche à exprimer une facette de la signification des programmes. Si dans la plupart des langages de programmation, la facette qui intéresse au premier plan est l'exécution du programme, il est tout à fait envisageable de trouver plusieurs facettes et de les exprimer dans différentes sémantiques ou encore de les composer à l'intérieur d'une même sémantique. Pour QML, deux significations peuvent être données aux contrats : une signification exécutoire consistant à vérifier si les valeurs mesurées dynamiquement selon les différentes dimensions observent les contraintes imposées par le contrat, ou encore une signification orientée typage consistant à vérifier statiquement si un contrat c_1 est conforme à un contrat c_2,

ce qui sera vrai si à chaque fois que le contrat c_1 est vérifié, le contrat c_2 l'est aussi. Dans cet exemple, nous avons retenu la première alternative.

Les domaines sémantiques de QML sont présentés à la figure 5. Le domaine caractéristique **V** des valeurs exprimables dans le langage est constitué d'identificateurs (dimensions énumération avec ou sans ordre), de nombres ou encore d'une liste d'identificateurs (dimensions ensembliste avec ou sans relation d'ordre). Une valeur de dimension est définie par le domaine **DimensionValue** qui est le produit d'une valeur exprimable et d'une unité de mesure. Les unités de mesure sont définies par une valeur de domaine fini indiquant l'absence d'unité (**none**), un pourcentage (**percent**) ou, si l'unité est nommée, une liste d'identificateurs représente les noms d'unités individuelles (la liste ⟨`m`, `sec`⟩ représentant l'unité composite m/sec).

Un type de contrat **ContractType** est un environnement liant des identificateurs de dimension à des types de dimension **DimensionType**. Un type de dimension est représenté par une fonction qui accepte un opérateur de contrainte (**ConstraintOperator**) et qui retourne une fonction prenant une valeur cible du domaine **DimensionValue** et retournant une fonction du domaine **ContractDimension**. Cette fonction va réaliser le test sur les valeurs mesurées selon cette dimension du contrat. On lui passe une valeur mesurée du domaine **DimensionValue** et elle retourne vrai ou faux selon que cette valeur respecte la contrainte ou non. Pour ce faire, elle vérifie la conformité des unités de mesure, puis elle appelle une autre fonction du domaine **DimensionTest** qui elle accepte une valeur exprimable **V** (sans unité) et retourne vrai si cette valeur respecte la contrainte sans égard à l'unité déjà vérifiée.

Bien sûr, comme nous le verrons dans les fonctions de valuation, la vérification tient compte de la sémantique de relation (**RelationSemantics**) associée à la dimension. Elle utilise la relation d'ordre définie sur la dimension (**OrderRelation**) lorsque nécessaire, définie comme une fonction prenant deux identificateurs et retournant vrai si la première est en relation avec la seconde, faux si la seconde est en relation avec la première et $\top$ s'il n'y a pas de relation définie entre les deux.

QML est essentiellement un langage permettant la création d'«ontologies» de qualité de service et donc d'environnements liant des identificateurs à des valeurs. Le premier niveau d'environnement créé par un programme QML est l'environnement global qui sert à lier les types de contrats, les contrats et les profils aux identificateurs qui leurs sont attribués dans les déclarations. Le second niveau est constitué des contrats et des types de contrats liant identificateurs à des types ou des contraintes pour former des dimensions. Le domaine **DV** est un domaine dit «caractéristique»[3] en ce qu'il indique le contenu des environnements dans le langage. En définissant **DV** comme la somme des domaines **TypedContract**, **ContractType**, **DimensionType** et **ContractDimension**, on indique la richesse des valeurs pouvant être liées à des noms dans le langage QML. Un tel domaine somme peut être vu comme introduisant une certaine «confusion» de types inappropriée en pratique (différents types de valeurs ne se retrouvant jamais dans les mêmes envronnements, pourquoi fusionner tout en un seul domaine ?), mais cette «confusion» sert ici à faire ressortir l'homogénéité sémantique de ces différentes valeurs. □

2.2.3 Fonctions de valuation

La troisième partie de la sémantique dénotationnelle est constituée des fonctions de valuation, qui indique comment calculer la dénotation d'une phrase d'un programme selon son domaine syntaxique. Ces fonctions de valuations se présentent sous la forme d'équations. Novices comme experts reconnaissent immédiatement ces équations à cause de leur utilisation très caractéristique des doubles crochets ($[\![\ ]\!]$). La raison d'être de ces doubles crochets est la séparation explicite entre le monde des domaines syntaxiques de celui des domaines sémantiques. Plus pragmatiquement, on notera que l'argument syntaxique d'une fonction de valuation est filtré sur patron, contrairement aux autres arguments portant des valeurs sémantiques qui sont normalement évalués avant l'appel, ce qui justifie cette distinction visuelle.

[3] En sémantique dénotationnelle, il est de pratique courante d'utiliser les mêmes noms d'une sémantique à l'autre de manière à faciliter la comparaison de la puissance d'expression des langages étudiés. Ainsi, le domaine **DV**, pour «denotable values», comprend les valeurs pouvant être désignées par des identificateurs dans le langage.

Domaines sémantiques :

υ	$\in$	**V**	=	$\mathbf{Ide} \oplus \mathbf{Num} \oplus \mathbf{Ide}^*$
μ	$\in$	**Unit**	=	$\{\bot, \mathbf{none}, \mathbf{percent}\} \oplus \mathbf{Ide}^*$
δ	$\in$	**DimensionValue**	=	$\mathbf{V} \otimes \mathbf{Unit}$
$\triangleright$	$\in$	**ConstraintOperator**	=	$\{\bot, \mathbf{equal}, \mathbf{lt}, \mathbf{ltequal}, \mathbf{gt}, \mathbf{gtequal}\}$
ψ	$\in$	**DimensionTest**	=	$\mathbf{V} \to \mathbf{T}$
Ψ	$\in$	**ContractDimension**	=	$\mathbf{DimensionValue} \to \mathbf{T}$
ω	$\in$	**Contract**	=	**Env**
Ω	$\in$	**TypedContract**	=	$\mathbf{Ide} \otimes \mathbf{Contract}$
σ	$\in$	**Set**	=	$\mathbf{Ide}^*$
φ	$\in$	**OrderRelation**	=	$(\mathbf{Ide} \otimes \mathbf{Ide}) \to (\mathbf{T} \oplus \mathbf{O})$
ς	$\in$	**POSet**	=	$\mathbf{Set} \otimes \mathbf{OrderRelation}$
κ	$\in$	**RelationSemantics**	=	$\{\bot, \mathbf{none}, \mathbf{increasing}, \mathbf{decreasing}\}$
ξ	$\in$	**DimensionSort**	=	$\mathbf{ConstraintOperator} \to (\mathbf{V} \to \mathbf{DimensionTest})$
ϕ	$\in$	**DimensionType**	=	$\mathbf{ConstraintOperator} \to$ $(\mathbf{DimensionValue} \to \mathbf{ContractDimension})$
Φ	$\in$	**ContractType**	=	**Env**
Δ	$\in$	**DV**	=	$\mathbf{TypedContract} \oplus \mathbf{ContractType} \oplus \mathbf{DimensionType}$ $\oplus \mathbf{ContractDimension}$
ρ	$\in$	**Env**	=	$\mathbf{Ide} \to (\mathbf{DV} \oplus \mathbf{T})$

FIG. 5 – Domaines sémantiques pour le langage QML

Pour chaque domaine syntaxique, une fonction de valuation différente est définie. On lui donne un nom, puis on la définit par une suite d'équations sémantiques. Il y a généralement une équation pour chacune des alternatives apparaissant dans la production (composite) de la syntaxe abstraite pour ce domaine syntaxique. Selon les besoins de la sémantique, et de chacun des domaines syntaxiques, les fonctions de valuation peuvent comporter un certain nombre d'arguments : environnement, mémoire, etc. Les équations définissent généralement des fonctions ; la notation la plus utilisée pour définir ces fonctions est le λ-calcul car cette notation possède les propriétés mathématiques nécessaires pour définir les valeurs des domaines sémantiques de fonctions [Sch86, Mos90].

Exemple. Fonctions sémantiques pour QML. La figure 6 présente les principales fonctions de valuation pour QML, c'est-à-dire celles qui concernent les domaines syntaxiques assurant les grands aiguillages dans le programme : programme, déclarations, déclarations de types de contrats et de contrats, etc. La dénotation associée à un programme par la fonction de valuation $\mathcal{P}$ est la paire (ρ, τ) où ρ est l'environnement obtenu par l'évaluation des déclarations et τ est le résultat de l'évaluation de l'expression de test. Les séquences de déclarations mènent à la construction d'un environnement par la fonction $\mathcal{DE}^*$, en appliquant à chaque déclaration la fonction $\mathcal{DE}$. Cette dernière appelle la fonction $\mathcal{CTD}$ s'il s'agit d'une déclaration de type de contrat et $\mathcal{CD}$ s'il s'agit plutôt d'une déclaration de contrat. La fonction $\mathcal{T}$ illustre l'utilisation des contrats. Avec l'identificateur de contrat i, on récupère le contrat dans l'environnement produit par $\mathcal{DE}^*$, puis avec l'identificateur de dimension i_1, on récupère la fonction de vérification associée à la dimension par le contrat (vu comme un environnement), qui est appliquée à la valeur mesurée *dv*.

La fonction $\mathcal{CTD}$ construit d'abord la dénotation du type de contrat puis crée une liaison avec son identificateur par *binding*, qui est combinée à l'environnement en cours de construction par *combine*. La dénotation de la séquence de déclaration de dimensions donnée par $\mathcal{CT}$ associe dans un environnement définissant le type de contrat chaque identificateur de dimension à la dénotation de la dimension. La fonction $\mathcal{D}^*$ traite la séquence de dimensions, chacune de celles-ci étant traitée par la fonction $\mathcal{D}$ traitant la déclaration du type de dimension par $\mathcal{DT}$ que nous expliquons plus loin. La fonction $\mathcal{CD}$ calcule la dénotation de la déclaration de contrat en appelant la fonction $\mathcal{CE}$, puis en liant son résultat à l'identificateur du contrat donné dans un environnement qui va être la dénotation de la déclaration de contrat. La fonction $\mathcal{CE}$ est également expliquée ci-après.

$\mathcal{P} :: [\![Program]\!] \to \mathbf{Env} \otimes \mathbf{T}$

$\mathcal{P}[\![\texttt{(program } d^* \ t \texttt{)}]\!] = \ \mathbf{let}\ \rho = \mathcal{DE}^*[\![d^*]\!]\rho_\emptyset\ \mathbf{in}\ (\rho, \mathcal{T}[\![t]\!]\rho)$

$\mathcal{DE}^* :: [\![Decl^*]\!] \otimes \mathbf{Env} \to \mathbf{Env}$

$\mathcal{DE}^*[\![d^*]\!]\rho = \ \mathbf{if}\ d^* = \langle\rangle\ \mathbf{then}\ \rho\ \mathbf{else}\ \mathcal{DE}^*[\![d^* \dagger 1]\!](\mathcal{DE}[\![d^* \downarrow_1]\!]\rho)\ \mathbf{endif}$

$\mathcal{DE} :: [\![Decl]\!] \otimes \mathbf{Env} \to \mathbf{Env}$

$\mathcal{DE}[\![\texttt{(type } ctd \texttt{)}]\!]\rho = \ \mathcal{CTD}[\![ctd]\!]\rho$

$\mathcal{DE}[\![\texttt{(contract } cd \texttt{)}]\!]\rho = \ \mathcal{CD}[\![cd]\!]\rho$

$\mathcal{T} :: [\![Test]\!] \otimes \mathbf{Env} \to \mathbf{T}$

$\mathcal{T}[\![\texttt{(test-verify } i\ i_1\ dv \texttt{)}]\!]\rho =$
$\quad \mathbf{let*}\ \omega = on_2(bound(\mathcal{I}[\![i]\!], \rho)\ |_{\mathbf{TypedContract}})$
$\quad \mathbf{and}\ \Psi = bound(\mathcal{I}[\![i_1]\!], \omega)\ |_{\mathbf{ContractDimension}}$
$\quad \mathbf{in}\ \Psi(\mathcal{DV}[\![dv]\!])$

$\mathcal{CTD} :: [\![ConTypeDecl]\!] \to \mathbf{Env} \to \mathbf{Env}$

$\mathcal{CTD}[\![\texttt{(contract-type-decl } i\ ct \texttt{)}]\!]\rho =$
$\quad \mathbf{let}\ \Phi = \mathcal{CT}[\![ct]\!]$
$\quad \mathbf{in}\ combine(binding(\mathcal{I}[\![i]\!], in\mathbf{DV}(\Phi)), \rho)$

$\mathcal{CT} :: [\![ConType]\!] \to \mathbf{ContractType}$

$\mathcal{CT}[\![\texttt{(contract-type } dim^* \texttt{)}]\!] = \ \mathcal{D}^*[\![dim^*]\!]\rho_\emptyset$

$\mathcal{D}^* :: [\![Dimension^*]\!] \otimes \mathbf{ContractType} \to \mathbf{ContractType}$

$\mathcal{D}^*[\![dim^*]\!]\rho = \ \mathbf{if}\ dim^* = \langle\rangle\ \mathbf{then}\ \rho\ \mathbf{else}\ \mathcal{D}^*[\![dim^* \dagger 1]\!](\mathcal{D}[\![dim^* \downarrow_1]\!]\rho)\ \mathbf{endif}$

$\mathcal{D} :: [\![Dimension]\!] \otimes \mathbf{ContractType} \to \mathbf{ContractType}$

$\mathcal{D}[\![\texttt{(dimension } i\ dt \texttt{)}]\!]\rho = \ combine(binding(\mathcal{I}[\![i]\!], in\mathbf{DV}(\mathcal{DT}[\![dt]\!])), \rho)$

$\mathcal{CD} :: [\![ConDecl]\!] \to \mathbf{Env} \to \mathbf{Env}$

$\mathcal{CD}[\![\texttt{(contract-decl } i\ ce \texttt{)}]\!]\rho = \ \mathbf{let}\ \omega = \mathcal{CE}[\![ce]\!]\rho\ \mathbf{in}\ combine(binding(\mathcal{I}[\![i]\!], in\mathbf{DV}(\omega)), \rho)$

FIG. 6 – Fonctions de valuation pour les domaines syntaxiques de QML assurant les grands aiguillages.

La fonction $\mathcal{DT}$ calcule la dénotation des déclarations de type de dimensions. Considérons uniquement le cas des dimensions simple (sans unité de mesure) :
$\mathcal{DT} :: [\![DimType]\!] \to \mathbf{DimensionType}$

$\mathcal{DT}[\![\texttt{(dimtype-simple } ds \texttt{)}]\!] =$
$\quad \mathbf{let}\ \xi = \mathcal{DS}[\![ds]\!]$
$\quad \mathbf{in}\ \lambda\triangleright.\lambda\delta.\ \mathbf{if}\ on_2(\delta) = (\mathbf{none}, \langle\rangle)$
$\qquad\qquad \mathbf{then}\ \lambda\delta_1.\ \mathbf{if}\ on_2(\delta_1) = on_2(\delta)$
$\qquad\qquad\qquad\qquad \mathbf{then}\ \xi(\triangleright)(on_1(\delta))(on_1(\delta_1))$
$\qquad\qquad\qquad\qquad \mathbf{else}\ \bot_{\mathbf{T}}$
$\qquad\qquad\qquad\qquad \mathbf{endif}$
$\qquad\qquad \mathbf{else}\ \bot_{\mathbf{ContractDimension}}$
$\qquad\qquad \mathbf{endif}$

On voit que le résultat est une fonction qui prend un opérateur de contrainte (qui sera défini dans le contrat), et qui retourne une fonction qui accepte une valeur de dimension cible et qui produira une fonction de vérification. La fonction sur la valeur cible commence par vérifier que son unité de mesure est bien «`none`», ce qui est le cas des dimensions simple. Si c'est le cas, elle retourne la fonction de vérification qui prend la valeur mesurée, vérifie son unité de mesure et si elle est conforme, elle appelle la fonction de vérification produite pour la sorte de dimension. Cette dernière fonction calculée en début de $\mathcal{DT}$ prend l'opérateur de contrainte, la valeur cible et la valeur mesurée, et elle retourne vrai si la contrainte est vérifiée et faux sinon. En voici la définition pour les dimensions énumération avec relation d'ordre :

$\mathcal{DS} :: [\![DimSort]\!] \rightarrow \mathbf{DimensionSort}$

$$
\begin{array}{l}
\mathcal{DS}[\![\texttt{(dimsort-enum-with } rs\ i^*\ o\texttt{)}]\!] = \\
\quad \textbf{let*}\ \sigma = \mathcal{I}^*[\![i^*]\!] \\
\quad \textbf{and}\ \kappa = \mathcal{RS}[\![rs]\!] \\
\quad \textbf{and}\ \varphi = \mathcal{O}[\![o]\!]\sigma \\
\quad \textbf{in}\ \lambda\triangleright.\lambda v.\ \textbf{if}\ (v\ |_{\mathbf{Ide}}\ \in \sigma)\ \wedge\ (\mathit{compatible}(\triangleright, \kappa)) \\
\qquad\qquad \textbf{then}\ \ \textbf{if}\ \triangleright = \textbf{equal} \\
\qquad\qquad\qquad \textbf{then}\ \lambda v_1.v = v_1 \\
\qquad\qquad\qquad \textbf{elseif}\ (\kappa = \textbf{increasing})\ \wedge\ (\triangleright = \textbf{gt}) \\
\qquad\qquad\qquad \textbf{then}\ \lambda v_1.v\ |_{\mathbf{Ide}} \prec_\varphi\ v_1\ |_{\mathbf{Ide}} \\
\qquad\qquad\qquad \textbf{elseif}\ (\kappa = \textbf{increasing})\ \wedge\ (\triangleright = \textbf{gtequal}) \\
\qquad\qquad\qquad \textbf{then}\ \lambda v_1.(v = v_1)\ \vee\ (v\ |_{\mathbf{Ide}} \prec_\varphi\ v_1\ |_{\mathbf{Ide}}) \\
\qquad\qquad\qquad \textbf{elseif}\ (\kappa = \textbf{decreasing})\ \wedge\ (\triangleright = \textbf{lt}) \\
\qquad\qquad\qquad \textbf{then}\ \lambda v_1.v_1\ |_{\mathbf{Ide}} \prec_\varphi\ v\ |_{\mathbf{Ide}} \\
\qquad\qquad\qquad \textbf{elseif}\ (\kappa = \textbf{decreasing})\ \wedge\ (\triangleright = \textbf{ltequal}) \\
\qquad\qquad\qquad \textbf{then}\ \lambda v_1.(v = v_1)\ \vee\ (v_1\ |_{\mathbf{Ide}} \prec_\varphi\ v\ |_{\mathbf{Ide}}) \\
\qquad\qquad\qquad \textbf{else}\ \perp_{\mathbf{T}} \\
\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad \textbf{else}\ \perp_{\mathbf{ContractDimension}} \\
\qquad\qquad \textbf{endif}
\end{array}
$$

Après avoir calculé l'ensemble des «valeurs» admissibles σ, la sémantique de relation κ et la relation d'ordre φ, on retourne une fonction qui prend un opérateur de contrainte et une valeur cible, et qui par analyse de cas retourne une fonction de vérification qui prendra une valeur mesurée et retournera vrai ou faux selon que la relation entre cette valeur et la valeur cible par l'opérateur de contrainte et vérifiée ou non. Un aspect sur lequel nous n'avons pas donné d'explication est celui de la compatibilité entre sémantique de relation et opérateur de contrainte. QML exige qu'une sémantique de relation `increasing` n'admet que des opérateurs de contrainte égal, plus grand ou plus grand ou égal, et inversement pour le cas `decreasing`, ce qui est vérifié par la fonction suivante :

$$
\begin{array}{l}
\mathit{compatible}(\triangleright, \kappa) = \\
\quad ((\kappa = \textbf{none})\ \wedge\ (\triangleright = \textbf{equal})) \vee \\
\quad ((\kappa = \textbf{increasing})\ \wedge\ ((\triangleright = \textbf{equal})\ \vee\ (\triangleright = \textbf{gt})\ \vee\ (\triangleright = \textbf{gtequal}))) \vee \\
\quad ((\kappa = \textbf{decreasing})\ \wedge\ ((\triangleright = \textbf{equal})\ \vee\ (\triangleright = \textbf{lt})\ \vee\ (\triangleright = \textbf{ltequal})))
\end{array}
$$

Côté contrats, les fonctions de valuation $\mathcal{CE}$ et $\mathcal{C}^*$ étant chargées de construire l'environnement liant nom de dimensions aux contraintes, nous nous contentons de donner la fonction $\mathcal{C}$ traitant les contraintes simples :

$\mathcal{C} :: [\![Constraint]\!] \rightarrow \mathbf{ContractType} \rightarrow \mathbf{Contract} \rightarrow \mathbf{Env} \rightarrow \mathbf{Contract}$

$$
\begin{array}{l}
\mathcal{C}[\![\texttt{(constraint-simple } i\ cp\ dv\texttt{)}]\!]\Phi\omega\rho = \\
\quad \textbf{let*}\ \delta = \mathcal{DV}[\![dv]\!] \\
\quad \textbf{and}\ \Delta = \mathit{bound}(\mathcal{I}[\![i]\!], \Phi) \\
\quad \textbf{and}\ \Delta_1 = \mathit{in}\mathbf{DV}(\Delta\ |_{\mathbf{DimensionType}}\ (\mathcal{CO}[\![cp]\!])(\delta)) \\
\quad \textbf{in}\ \mathit{combine}(\omega, \mathit{binding}(\mathcal{I}[\![i]\!], \Delta_1))
\end{array}
$$

Cette fonction calcule dans δ la valeur cible de la contrainte, puis récupère dans le type de contrat Φ la valeur Δ associée à la dimension. Cette valeur est une fonction calculée par $\mathcal{DT}$ que l'on récupère par projection sur le domaine **DimensionType** et que l'on applique à l'opérateur de contrainte pour obtenir la fonction génératrice de fonction de vérification à laquelle on passe la valeur cible δ. La fonction de vérification ainsi obtenue, est injectée dans le domaine **DV** pour obtenir Δ_1 qui sera placée dans l'environnement qui va représenter le contrat.

3 Définition du modèle UML/OCL

3.1 Modélisation de la syntaxe abstraite

La modélisation de la syntaxe abstraite est certainement la partie la mieux connue de l'approche proposé ici. En effet, la grammaire abstraite décrivant des arborescences, ces arborescences peuvent être représentées sous forme d'objets. L'idée est de définir une classe pour chaque domaine syntaxique. Lorsqu'un domaine syntaxique possède plusieurs alternatives, sa classe est définie comme abstraite et on définit autant de sous-classes concrètes qu'il y a d'alternatives dans la production. Cette approche est bien connue et a été capturée par une partie du patron de conception *Interpréteur* [GHJV95, p. 243]. Il est évident que l'application systématique de ce patron produit des modèles relativement lourds et souvent moins faciles à manier par l'humain que les grammaires abstraites corrspondantes. L'avantage ici d'avoir un modèle vient plutôt du fait qu'il peut être manipulé automatiquement par le modeleur.

Exemple. Modélisation de la syntaxe abstraite de QML. La figure 7 présente un diagramme de classes partiel de la syntaxe abstraite de QML se fondant sur ce patron. Pour ne pas encombrer inutilement le diagramme déjà lourd, nous n'avons pas inclus les relations entre les différentes sortes de dimension et la classe `Identifier` qui représentent le fait que les déclarations de sortes de dimensions introduisent les noms des différentes valeurs admises par la dimension.

3.2 Modélisation des domaines sémantiques

Notre objectif est de transposer en termes d'objets la sémantique formelle de vérification présentée dans la section précédente. Le passage de la sémantique dénotationnelle à un modèle objet est tout sauf direct. En effet, le λ-calcul, ossature de la partie calculatoire de la sémantique dénotationnelle, tire une grande partie de sa puissance d'une utilisation systématique des fonctions d'ordre supérieur, beaucoup moins usitées dans l'approche objet. *A contrario*, la modélisation objet s'appuie sur l'héritage, le polymorphisme et, éventuellement, la surcharge des méthodes en fonction des types et du nombre de leurs arguments.

Ainsi, tirer une modélisation objet d'une sémantique dénotationnelle demande de définir une hiérarchie d'héritage pour représenter les données, de répartir les fonctions utilisant de la sélection sur le type des données en méthodes sur les classes correspondantes, et enfin d'utiliser la composition de méthodes par super ou par envoi de messages là où la sémantique dénotationnelle compose des fonctions.

Plus spécifiquement, les heuristiques suivantes guident le passage des domaines sémantiques aux objets :

1. établissement d'une correspondance entre les domaines de base, comme **Ide**, **Num** et **T** et les types de données du langage de modèlisation, en l'occurence UML/OCL ;
2. modélisation des domaines sommes par une hiérachie d'héritage où le domaine somme est superclasse de ses composants, eux-mêmes modélisés par des classes représentant les étiquettes de ces derniers ;
3. modélisation des domaines produits par des classes faisant l'agrégation des composants par un nombre de variables d'instance correspondant au nombre de composants du domaine produit,
4. utilisation de domaines énumération pour représenter les domaines finis (comme **ConstraintOperator** et **RelationSemantics**).

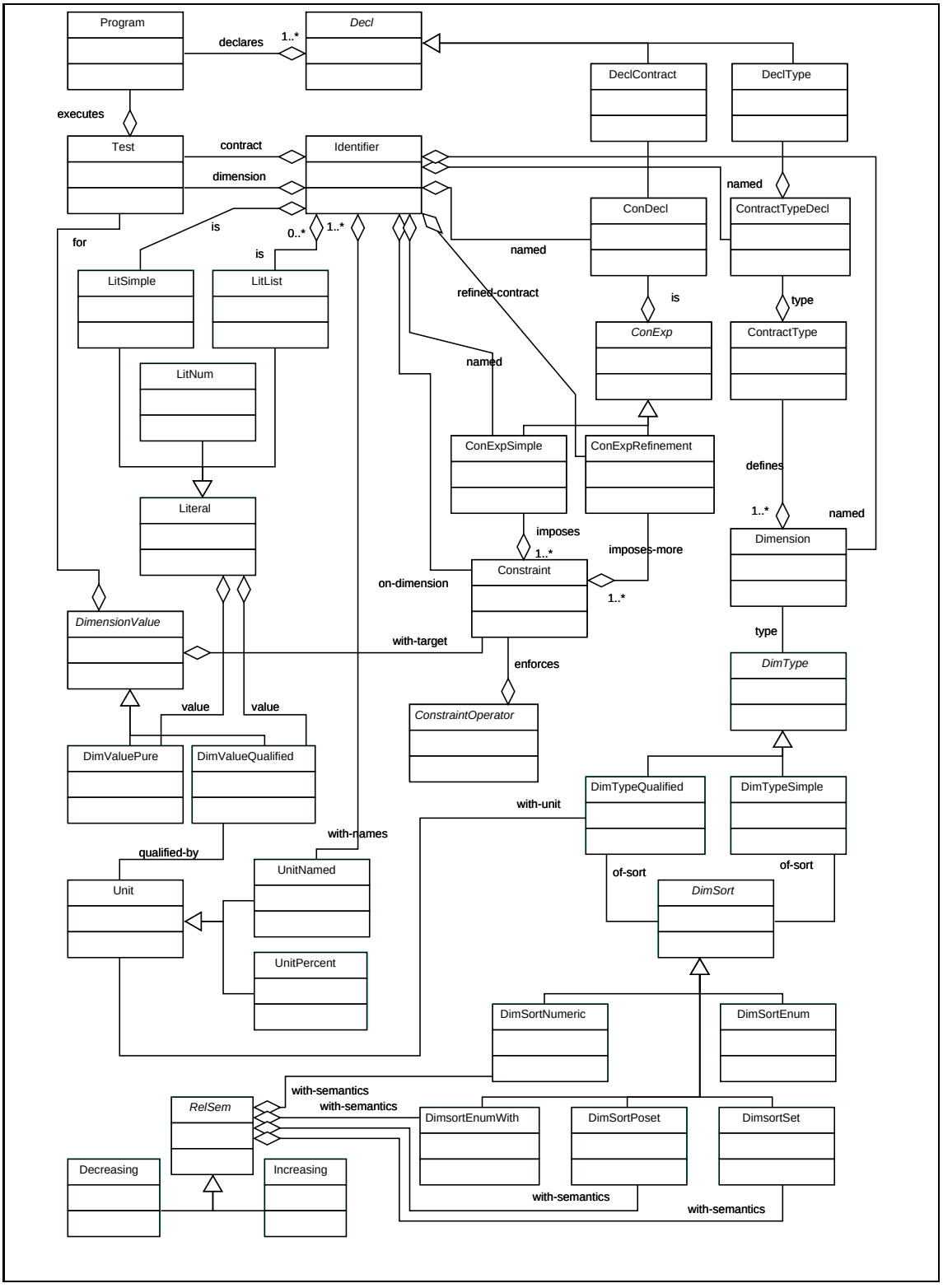

FIG. 7 – Modélisation de la syntaxe abstraite de QML.

Exemple. Les domaines sémantiques de QML. Pour ce qui concerne les domaines sémantiques que nous avons définis pour QML (figure 5), les domaines standards, comme **Ide**, **Num** et **T**, sont directement modélisés par les types chaînes de caractères (`String`), nombres (ici `Real`) et booléens (`Boolean`), que nous suppposerons connus du modèle (ce sont des types de base d'OCL). Les domaines finis **ConstraintOperator** et **RelationSemantics** se prêtent à une modélisation en type énumération. Ils sont définis par :

`ConstraintOperator` = `Enum {equal, lt, ltequal, gt, gtequal}`

`RelationSemantics` = `Enum {none, increasing, decreasing}`

Le domaine somme **V** est représenté par une classe abstraite `V` avec trois sous-classes concrètes pour les sommandes nommées `StringOfV`, `RealOfV` et `StringsOfV`. Le domaine **Env** est représenté par la classe `Env` liant des chaînes de caractères à des valeurs dénotables du domaine **DV** ici représenté par la classe abstraite `DV`. La classe `DV` a quatre sous-classes pour les sommandes du domaines : `DimensionTypeOfDV`, `ContractDimensionOfDV`, `TypedContractOfDV` et `ContractTypeOfDV`. Chacune de ces classes contient une valeur du type correspondant de la somme, c'est-à-dire `DimensionType`, `ContractDimension`, `TypedContract` et `ContractType`. Un contrat typé du domaine **TypedContract** est défini par la classe `TypedContract` comportant deux valeurs, le nom du type de contrat (`String`) et le contrat lui-même défini par la classe `Contract`. Les domaines **Contract** et **ContractType** étant des sortes d'environnement, nous avons défini les classes correspondantes comme sous-classes de la classe `Env`. Les domaines **DimensionType**, **ContractDimension**, **DimensionSort** et **DimensionTest** sont des domaines de fonctions qui sont simplement représentés par des classes de même noms ici. Nous revenons immédiatement après sur le contenu de ces classes. □

Pour la modélisation des domaines fonctionnels, outre l'évidente utilisation de méthodes pour les représenter, des difficultés se posent en ce qui concerne les fonctions d'ordre supérieur et les fonctions (souvent anonymes) utilisées comme entités de plein droit. Les principales heuristiques pour passer des fonctions de valuation à la modélisation objet sont :

1. modélisation des fonctions comme entités de plein droit par des représentation objet de fermetures ;
2. utilisation de variables d'instance des objets fermeture pour capturer les arguments des fonctions curryfiées ;
3. utilisation de la surcharge sur les types des arguments lorsqu'une fonction est définie sur des arguments de types sommes (autre que celui du receveur) ;
4. introduction de l'héritage là où cette relation existe entre les domaines de manière à discriminer entre les types de valeurs (là où on utilise souvent des fonctions comme entités de plein droit en sémantique dénotationnelle) ;
5. utilisation de préconditions sur les méthodes pour exprimer la sémantique statique ;
6. utilisation de postconditions pour exprimer la sémantique dynamique, en remarquant que l'on peut toujours écrire une postcondition établissant à quoi doit être égal le résultat d'une méthode pour capturer le code de la méthode ;

Exemple. Domaines de fonctions de la sémantique de QML. Les quatre domaines de fonctions centraux de notre sémantique de QML sont les domaines **DimensionType**, **ContractDimension**, **DimensionSort** et **DimensionTest** que nous avons représentés par des classes de même nom. Dans la sémantique, ce sont des fonctions de vérifications et leurs rôles respectfs sont les suivants. La fonction du domaine **DimensionTest** se borne à vérifier qu'une valeur mesurée respecte la contrainte posée sur une dimension d'un contrat. Cette fonction est produite par une fonction curryfiée du domaine **DimensionSort** qui accepte un opérateur de contrainte et une valeur cible de la contrainte, et retourne une fonction du domaine **DimensionTest**. Les fonctions du domaine **ContractDimension** acceptent une valeur cible et une valeur mesurée du domaine **DimensionValue**. Elles vérifient la conformité de leurs unités de mesure puis elles appellent la fonction du domaine **DimensionSort** pour propager la vérification sur la valeur du domaine **V**, débarrassée de son unité de mesure déjà vérifiée. Enfin, les fonctions du domaine **DimensionType**

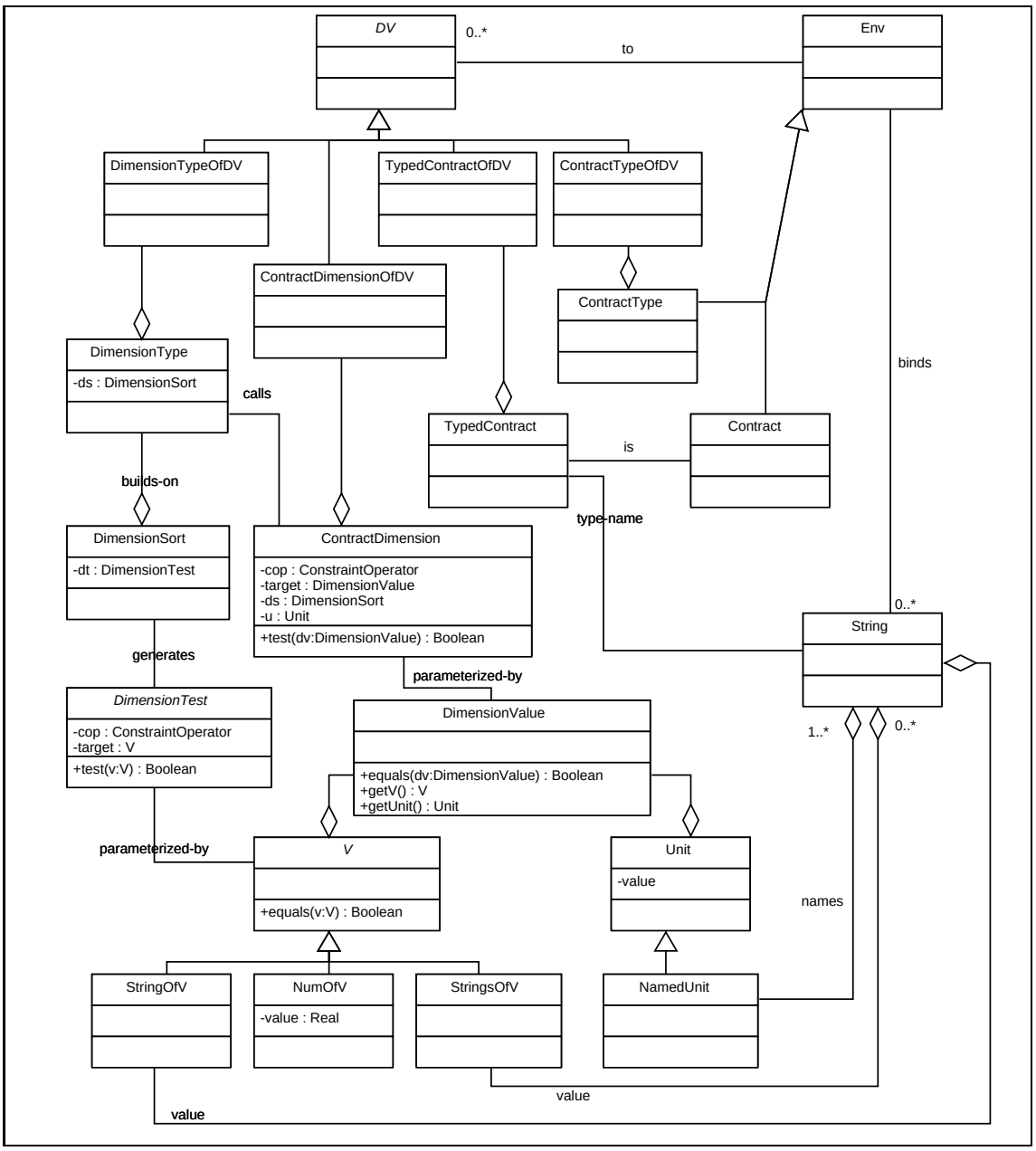

FIG. 8 – Modélisation des domaines sémantiques de QML.

enveloppent le tout en construisant d'une part les fonctions **DimensionSort** et d'autre part celles du domaine **ContractDimension**.

Pour représenter sous forme d'objets ces fonctions, nous utilisons les deux premières heuristiques précédentes : chaque classe est dotée d'une méthode appelée `test`, avec les paramètres appropriés dans chaque cas. Les méthodes des classes `DimensionType` et `ContractDimension` prennent en paramètres opérateur de contrainte et valeur cible pour initialiser les variables d'instances correspondantes des classes `DimensionSort` et `DimensionTest`. Les vérifications de conformité sont faites au plus tôt et le résultat est vrai si les valeurs respectent la contrainte et faux sinon.

```
DimensionTypeSimple::test(c:ConstraintOperator,
       t:DimensionValue, m:DimensionValue):Boolean
pre  : self.values.member(t.getV()) and compatible(c,self.rs)
post : cd.getCop() = c and cd.getTarget() = t and cd.getDs() = self.ds
post : result = cd.test(m)
```

On vérifie en précondition que la valeur cible fait partie des valeurs admissibles de la dimension et que l'opérateur de contrainte est compatible avec la sémantique de relation choisie. La première postcondition établit que les variables d'instances de l'objet `ContractDimension` associé au type sont bien définies (on suppose l'existence des accesseurs appropriés, non-mentionnés dans le diagramme pour ne pas l'alourdir inutilement), et la seconde établit le résultat comme l'appel à la méthode `test` de cette dimension de contrat.

```
ContractDimension::test(dv:DimensionValue):Boolean
pre  : self.u = dv.getUnit()
post : result = self.ds.test(dv.getV())
```

Ici, c'est l'exactitude de l'unité de mesure de la valeur mesurée qui est vérifiée en précondition, alors que la postcondition s'en remet à la sorte de dimension pour faire la vérification sur la valeur débarrassée de son unité de mesure.

```
DimensionSort::test(c:ConstraintOperator, t:V, m:V):Boolean
pre  : self.values.member(t) and compatible(c,self.rs)
post : dt.getCop() = c and dt.getTarget() = t
post : result = dt.test(m)
```

La partie valeur (sans unité) de la cible et de la valeur mesurée sont ici vérifiées respectivement en pré- et en postcondition.

```
EWDimensionTest::test(v:V):Boolean
pre  : self.values.member(self.target) and compatible(self.cop,self.rs)
post : self.cp = #equal implies result = (self.target = v)
post : (self.rs = #increasing and self.cp = #gt)
          implies result = orel.areInRelation(self.target,v)
post : (self.rs = #increasing and self.cp = #gtequal)
          implies result = (self.target = v or orel.areInRelation(self.target,v)
post : (self.rs = #decreasing and self.cp = #lt)
          implies result = orel.areInRelation(v,self.target)
post : (self.rs = #decreasing and self.cp = #ltequal)
          implies result = (self.target = v or orel.areInRelation(v,self.target)
```

Ici, la postcondition montre le calcul à faire pour vérifier la valeur mesurée en utilisant simplement des contraintes d'égalités. Les alternatives de la fonction sont devenues une analyse de cas par des implications en OCL.

Notons l'utilisation des deux dernières heuristiques, celles des pré- et postconditions, pour définir la sémantique statique (vérification de conformité des opérateurs et sémantique de relation de même que l'appartenance de la valeur cible aux valeurs admissibles sur la dimension) et la sémantique dynamique (vérification du respect de la contrainte par la valeur mesurée). Enfin, les fonctions des domaines **DimensionType** et **DimensionTest** différent d'une sorte de dimension à l'autre, d'où l'utilisation de l'heuristique sur l'héritage par introduction de définitions différentes sur des sous-classes appropriées, comme ici `DimensionTypeSimple` pour les dimensions sans unités de mesure et `EWDimensionTest` pour les sortes de dimension énumération avec relation ordre. On suppose, dans la défintiion de leurs méthodes `test`, que ces classes possèdent une variable d'instance `values` contenant l'ensemble de valeurs admissibles selon ladite dimension. □

Certaines dénotations qui reviennent régulièrement dans les sémantiques dénotationnelles méritent un traitement particulier. C'est le cas des environnements. Ils ont généralement une représentation fonctionnelle dans les sémantiques dénotationnelles. En passant à une modélisation objet, il peut être plus intéressant d'identifier des entités qui jouent le rôle d'environnement de manière à plonger les valeurs sémantiques dans ces entités. Un objet joue naturellement le rôle d'environnement, puisqu'il sert à lier des identificateurs (variables d'instance, variables de classe ou méthodes) à des valeurs. Il est possible d'utiliser cette analogie de manière à représenter plu-

sieurs domaines sémantiques environnements par des objets dont la classe déclarera les variables ou méthodes du nom des identificateurs à lier. Voir notre étude sémantique de QML pour plus de détails sur cette approche [Mal02].

3.3 Modélisation des fonctions de valuation

Pour ce qui concerne les fonctions de valuation, l'idée générale consiste à les transformer en méthodes définies sur les classes de la syntaxe abstraite de manière à produire des instances des classes représentant les valeurs sémantiques concernées. On retrouve ici les idées du patron *Interpréteur* déjà mentionné, à ceci près qu'en lieu et place des fonctions d'évaluation d'un interpéteur, nos méthodes réalisent les fonctions de valuation. Comme il s'agit de création d'objets, on produira pour l'essentiel un entrelacement de méthodes de création d'objets (constructeurs en Java) s'appelant les unes les autres.

Exemple. Fonctions de valuations de QML. Par manque d'espace, il ne nous est pas possible ici de détailler la réalisation de toutes les fonctions de valuation. Prenons un seul exemple, celui de la fonction de valuation pour les sortes de dimensions énumération avec relation d'ordre. Si les fonctions de valuations sont appelées `val`, on définirait :

```
DimSortEnumWith::val():DimensionSort
post : result = new DimensionTest(self.values, relsem.val(), o.val())
DimensionSort::DimensionSort(is :Sequence(String),
       rs1 :RelationSemantics, or1 :OrderRelation
post : self.values = is and self.rs = rs1 and self.or = or1
```

Nous supposons ici que la classe `DimSortEnumWith` possède des variables d'instance `values`, `relsem` et `o` contenant respectivement les nœuds de types `Sequence(String)`, `RelSem` et `Ordre` des sous-expressions. Nous supposons également que la fonction de valuation sur les chaînes de caractères (représentant à la fois le domaine syntaxique *Identifier* et le domaine sémantique **Ide**) est l'indentité. Enfin nous supposons que l'égalité sur les valeurs des domaines est défini correctement, c'est-à-dire qu'elle représente bien l'égalité profonde et non superficielle là où il le faut. □

4 Travaux connexes

Il n'existe pas à notre connaissance de tentative de modélisation en UML des sémantique s dénotationnelles de langages de programmation. Par contre, l'approche axiomatique a déjà été utilisée, merci au pré- et postconditions en OCL, dont l'exemple le plus complet est celui de l'*Action Semantics* pour UML [OMG01]. Nous avons également mentionné à quelques reprises la parenté de notre approche de modélisation des sémantique dénotationnelles avec celle proposée par le patron de conception *Interpréteur* tel que présenté par Gamma et al. [GHJV95, p. 243].

5 Conclusions, évaluation et perspectives

Nous avons proposé une approche systématique et heuristique pour modéliser en UML et OCL des sémantiques dénotationnelles des langages de programmation. Les deux avantages de cette approche sont : une expression de la sémantique des langages plus abordable par les programmeurs tout en étant fondée sur une approche formelle, et une intégration à la MDA de modèles des langages de programmation qui utilise UML, sorte de *lingua franca* des architectures fondées sur les modèles. Nous avons illustré cette approche par son application à la sémantique d'un langage spécifique au domaine des contrats de qualité de service appelé QML.

Bien que plus abordable à la majorité des programmeurs qu'une sémantique dénotationnelle, le modèle UML/OCL de la sémantique de QML demeure complexe. Sur certains points, comme la syntaxe abstraite, l'utilité d'avoir un modèle UML ne se révèle que lorsqu'il s'agit de le traiter automatiquement par des outils UML ; sinon, la syntaxe abstraite est probablement plus facile à lire. La partie transformation des domaines de fonctions se repose sur des heuristiques plus floues

que les autres parties. Il s'agit ici d'exercer encore une fois l'intuition et l'expérience du modélisateur de manière à introduire des modèles ayant à la fois uns filiation claire avec la sémantique dénotationnelle et qui utilise bien les possibilités de la modélisation objet.

L'exercice auquel nous nous sommes livrés n'est bien sûr pas complet. Les heuristiques que nous avons proposés s'appliquent bien à des sémantiques ne comportant que des objets mathématiques relativement simples, et en fait proches des structures de données classiques. Il resterait à explorer plus systématiquement le catalogue des domaines sémantiques connus de manière à en proposer des traduction en terme de modèle objet pour compléter ce travail.

Remerciements

Je tiens à remercier les membres du projet Triskell de l'IRISA pour le soutien apporté et les discussions autour de la sémantique de QML dont sont issus les idées présentées ici. Que Christian Queinnec reçoive également mes remerciements pour son génial LiSP2TEX qui m'a servi à mettre en page la sémantique dénotationnelle. Les modèles UML ont été construit sous Poseidon 1.0, avec les avantages et les défauts de cet outil.

Références

[dBdV96] J. de Bakker et E. de Vink. *Control-Flow Semantics.* MIT Press, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Mal02] J. Malenfant. Une étude sémantique du langage QML. Rapport technique, IRISA, 2002.

[Mos90] P.D. Mosses. Denotational Semantics. In *Handbook of Theoretical Computer Science*, chap. 11, pages 575–631. Elsevier Science Publishers & MIT Press, 1990.

[OMG01] OMG. *Action Semantics for the UML*, March 2001.

[Sch86] D.A. Schmidt. *Denotational Semantics : a methodology for language development.* Wm. C. Brown Publishers, 1986.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5 :522–587, 1976.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings International Colloquium on Automata, Languages, and Programming '82*, 1982.

[Sto77] Joseph E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399